10/01/2024
Wednesday

* **Paint the fence -**

There are n fences which can be coloured with one of the k colours in such a way that not more than two adjacent fences have the same colour.

I/P - n = 4, k = 3
O/P - 66

- **Code using recursion**

```
int solveUsingRec(int n, int k) {

    //base case
    if(n==1) return k;
    if(n==2) return k + (k*(k-1));

return (solveUsingRec(n-1,k) + solveUsingRec(n-2,k))*(k-1);
}

int main() {

    int n=4;
    int k=3;
    int ans = solveUsingRec(n,k);
    cout << ans<<endl;
}
```

- Code using recursion + memoization

```
int solveUsingMemo(int n, int k, vector<int>& dp){

    // base case
    if (n==1) return k;
    if (n==2) return k + (k*(k-1));
    // if ans already exists,
    if (dp[n] != -1){
        return dp[n];
    }
// store & return using dp array
dp[n]=(solveUsingMemo(n-1,k,dp) + solveUsingMemo(n-2,k,dp))
                                        *(k-1);

 return dp[n];
}

int main(){

    int n=4;
    int k=3;
    // create dp array
    vector<int>dp(n+1,-1);
    int ans= solveUsingMemo(n,k,dp);
    cout << ans << endl;
}
```

- Code using tabulation

```
int solveUsingTabu(int n, int k){
    // create dp array
    vector<int> dp(n+1, -1);

    // analyse base case & fill initial array
    dp[1] = k;
    dp[2] = k + (k*(k-1));

    // fill remaining array
    for(int i=3; i<=n; i++){
        dp[i] = (dp[i-1] + dp[i-2]) * (k-1);
    }
    return dp[n];
}

int main(){
        int n = 4;
        int k = 3;

        int ans = solveUsingTabu(n,k);
        cout << ans << endl;
}
```

→ As we can see here that dp[i] depends only on its two previous answers i.e. dp[i-1] and dp[i-2]. So, we can do space optimisation here by using two variables which will store answer of two previous indices.

- **Space optimisation in Paint the fence problem -**

```
int solveUsingSO(int n, int k){

    int prev2 = k;
    int prev1 = k + (k*(k-1));
    int curr;

    for(int i=3; i<=n; i++){
        curr = (prev2 + prev1)*(k-1);

        //shifting
        prev2 = prev1;
        prev1 = curr;
    }
    return curr;
}

int main(){
    int n=4;
    int k=3;

    int ans = solveUsingSO(n, k);
    cout << ans << endl;
}
```

Note - Always remember to shift the variables while doing space optimisation in code.

* **0/1 Knapsack problem -**

Given a knapsack which can hold w weight units. We have a total of n items to chose from, whose values are represented by an array val[] and weights represented by an array wt[].

We can either include an item in our knapsack or exclude it, but not include a fraction of it. Also, can't include an item multiple times.

I/P - N = 3 (No. of items)
W = 50 (capacity of knapsack)
values[] = {60, 100, 120} (value/profit per item holds)
weight[] = {10, 20, 30} (Weight per item)

O/P - 220

The problem is to determine the number of each item to include in a collection so that total weight is less than or equal to a given limit or capacity of knapsack and the total value/profit is as large as possible.

- Code using recursion

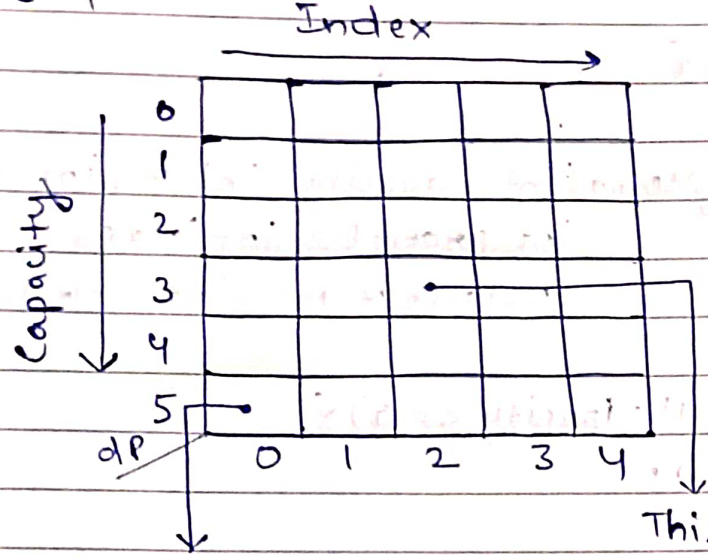
```
class Solution {
public:

int solveUsingRec(int capacity, int n, int wt[],
                  int profit[], int index){
    //base case
    if(index >= n || capacity <= 0){
        return 0;
    }

    //include an item
    int include = 0;
    if(wt[index] <= capacity){
        include = profit[index] + solveUsingRec(capacity -
                                  wt[index],
                  n, wt, profit, index+1);

    //exclude an item
    int exclude = 0 + solveUsingRec(capacity, n, wt, profit,
                                    index+1);

    return max(include, exclude);
}

int knapsack(int W, int wt[], int val[], int n){

    int index = 0;
    return solveUsingRec(W, n, wt, val, index);
}
};
```

Note - Here, two arguments are changing for every function call i.e. capacity and index. So, we will use 2-D dp here.

- Code using recursion + memoization

```
class Solution {
public:
int solveUsingMemo(int capacity, int n, int wt[],
                   int profit[], int index,
                   vector<vector<int>>& dp){
 // base case
 if(index >= n || capacity <= 0){
     return 0;
 }
 // if ans already exists,
 if(dp[capacity][index] != -1){
     return dp[capacity][index];
 }
 // include an item
 int include = 0;
 if(wt[index] <= capacity){
  include = profit[index] + solveUsingMemo(capacity -
                                wt[index],
                                n, wt, profit, index+1,
                                dp);
 }
 // exclude an item
 int exclude = 0 + solveUsingMemo(capacity, n, wt, profit,
                                  index+1, dp);
 // store & return using dp array
 dp[capacity][index] = max(include, exclude);
 return dp[capacity][index];
}
int knapsack(int W, int wt[], int val[], int n){
    int index = 0;
    vector<vector<int>>dp(W+1, vector<int>(n+1, -1));
    return solveUsingMemo(W, n, wt, val, index, dp);
}
};
```

let the capacity be 5 and number of items be 4



Final answer will always formed here i.e. when capacity equals capacity, and index equals 0 i.e. at dp[capacity][0].

This cell here represents that for capacity = 3 and index = 2, the stored value is the answer. Similar with others.

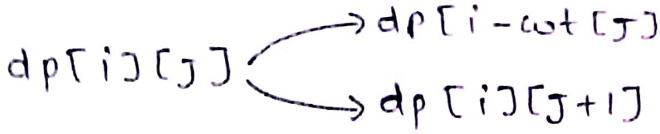
⇒ In the memoization code, we are moving —
for capacity = (capacity → 0)
for indices = (0 → index)

⇒ That's why, we will move the exact opposite for the tabulation —
for capacity = (0 → capacity)
for indicies = (index → 0)

- Code using tabulation

```
class Solution {
public:
int solveUsingTabulation(int capacity, int n, int wt[],
                         int profit[]) {

  // create dp array
vector<vector<int>> dp(capacity+1, vector<int>(n+1,-1));

  // maintain initial state of array
 for(int i=0; i<=capacity; i++) {
      dp[i][n] = 0;
 }

  // fill remaining array
 for(int i=0; i<= capacity; i++) {
   for(int j=n-1; j>=0; j--){
     int include=0;
      if(wt[j]<=i){
         include = profit[j] + dp[i-wt[j]][j+1];
      }
     int exclude = 0 + dp[i][j+1];

      dp[i][j] = max(include, exclude);
    }
   }
 return dp[capacity][0];
}
int knapsack( int W, int wt[], int val[], int n){

    return solveUsingTabulation(W, n, wt, val);
 }
};
```

Note - As we can see in the tabulation approach,



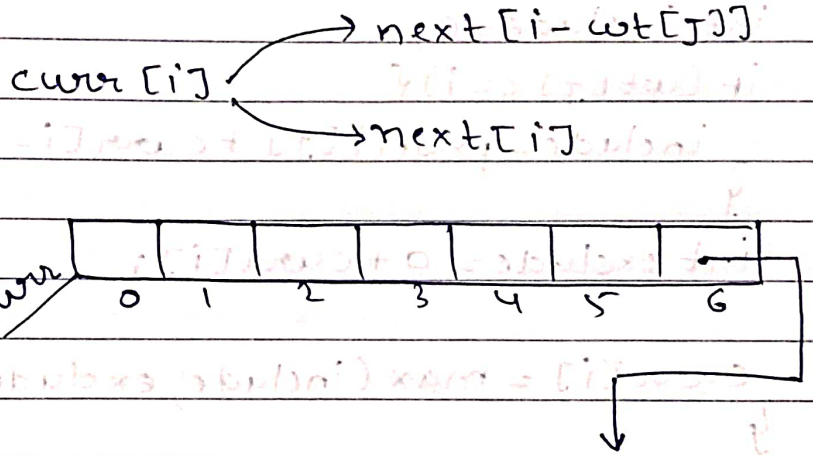
The answer of dp[i][j] always depends upon the column next to it. So, rather than using 2-D array, we can use 2 1-D arrays for space optimisation.

- Space-optimised code -

```
class Solution{
public:
int solveUsingSO(int capacity, int n, int wt[], int profit[])
{
    vector<int> next(capacity+1, 0);
    vector<int> curr(capacity+1, 0);

for(int j=n-1; j>=0; j--){
    for(int i=0; i<=capacity; i++){
        int include=0;
        if(wt[j]<=i){
          include = profit[j] + next[i-wt[j]];
        }
        int exclude= 0+ next[i];

        curr[i] = max(include, exclude);
    }
    //shifting
     next = curr;
  }
  return curr[capacity];
}
int knapsack(int W, int wt[], int val[], int n){

    return solveUsingSO(W, n, wt, val);
  }
};
```

⇐ In tabulation approach, we are building the answer row-wise.
Here, in space optimised code, we are building the answer column-wise. That's why loops are interchanged.

⇐ Now, in the previous one, we are using two vectors. We can optimise it more by using a single vector.
In the previous space optimised code -



let for index 6, the answer depends upon next[6] and next[6-wt[j]].
The next[6-wt[j]] will always be found in its previous indices.
Rather than, storing the answer in another array, we will use that array only.
for that, previous space optimised code mein, capacity wali loop ko opposite direction chla denge.

- **More space-optimised code -**

```
class Solution{
public:
int solveUsingSO2(int capacity, int n, int wt[], int priofit[])
{
   vector<int> curr(capacity+1, 0);

 for (int J=n-1; J>=0; J--){
   for (int i= capacity; i>=0; i--){
        int include=0;
        if (wt[J] <= i){
          include= priofit[J] + curr[i-wt[J]];
        }
        int exclude= 0+curr[i];

        curr[i] = max(include, exclude);
       }
    }
  return curr[capacity];
}
int knapSack(int W, int wt[], int val[], int n){

    return solveUsingSO2(W, n, wt, val);
  }
};
```

Note - 2-D dp mein generally yhi scenerio hoga, space optimisation ke liye 2-D vector se 2 1-D vectors pr aayenge, then if possible to single vector se krenge for more space optimised solution.